# Exercise 2

### Assignment 1: **Replace the activation function with the Sigmoid function. Train the network, and compare with the original activation function. What differences did you experience? Why do you think this difference happened?**

The Sigmoid function is:



And derivative of the Sigmoid function is:



In Python one can implement sigmoid function with:

return np.exp(np.fmin(x, 0)) / (1 + np.exp(-np.abs(x)))

But I have used scipy.special.expit(x) because it’s more secure and effective.

So it looks like this:

from scipy import special

def sigmoid(x):

return special.expit(x)

Derivative looks like this:

def sigmoid\_grad(x):

return sigmoid(x)\*(1 - sigmoid(x))

I try with epoch 10,000 and learning rate 0.6, and I did print the value of last error.

I have made some changes to the code, so the error doesn’t show squared error of last sample (last row of data), but mean of squared error of all rows in an iteration. This is how the code looks like (changes marked in red):

errors\_list = list()

objective = list()

descent\_list\_1 = list()

descent\_list\_2 = list()

ws\_list\_1 = list()

ws\_list\_2 = list()

error\_list = list()

for epoch in range(epochs): #number of training iterations, or times to change the weights of the nn

for i in range(X.shape[0]): #for all samples in X, each streetlight

layer\_in = X[i:i+1]

#forward pass/prediction

layer\_1 = relu(layer\_in.dot(ws\_1))

layer\_out = layer\_1.dot(ws\_2)

#calc error/distance (how far are we from goal)

delta\_2 = layer\_out - y[i:i+1]

#calc the the error each node in prev layer contributed

delta\_1 = delta\_2.dot(ws\_2.T) \* relu\_grad(layer\_1)

descent\_2 = (layer\_1.T.reshape(hidden\_nodes,1).dot(delta\_2))

descent\_1 = (layer\_in.T.reshape(X.shape[1],1).dot(delta\_1))

#update weights

ws\_2 -= lr \* descent\_2

ws\_1 -= lr \* descent\_1

error = delta\_2\*\*2

error\_list.append(error[0][0])

if epoch % 1 == 0:

objective.append(sum(error\_list) / X.shape[0])

error\_list = list()

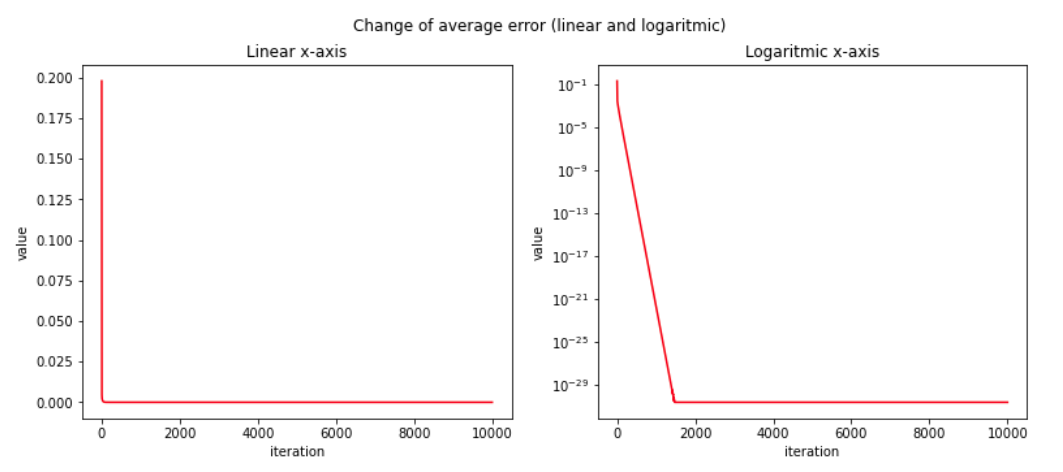
descent\_list\_2.append(descent\_2.copy())

descent\_list\_1.append(descent\_1.copy())

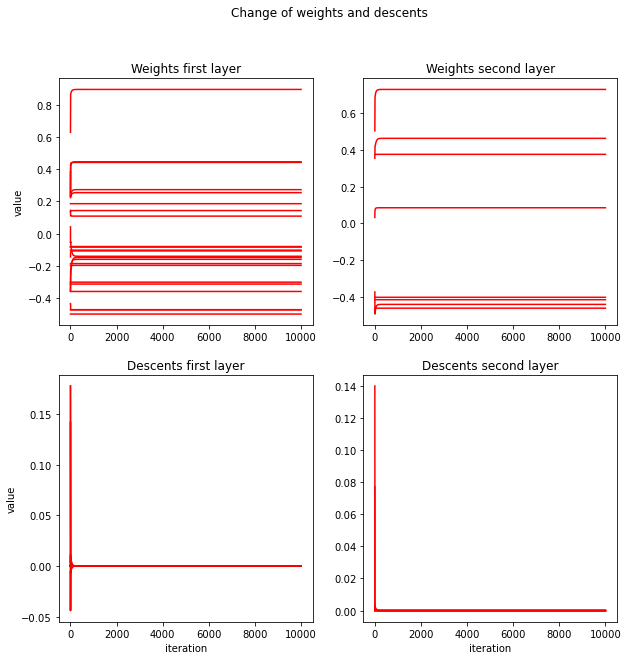
ws\_list\_1.append(ws\_1.copy())

ws\_list\_2.append(ws\_2.copy())

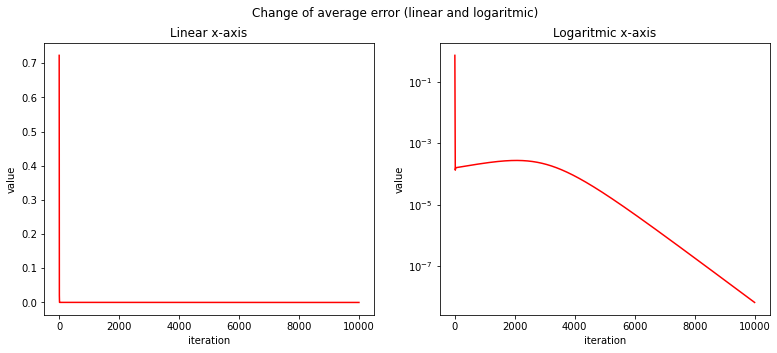
With ReLU the average error reduces quickly to 8.37786197e-34 and stays there. The graph below shows how the average error reduces:



And the weights converge quickly:



But when I use Sigmoid the error reduces more slowly:



And weights converge more slowly too.

Why is that?

Values of the input data is somewhere betwen 1 and 0.

ReLU function is non-saturating function while sigmoid is a staurating function. What does it mean:

It means ReLU(x) = ∞ when x = ∞, and derivative of ReLU is either 1 or 0.

But Sigmoid(x) converges to 1 when x = ∞ and 0 when x = -∞

And the derivative of Sigmoid(x) converges to 0 when x = ∞ or x = -∞

The derivative of the activation function is a factor of the gradient descent, also it determines how much value of the weights do change.

Since derivative of sigmoid reaches 0 in some activations, the gradient descent reduces in value, so the change in weights also reduces when the error reduces. But derivative of ReLU is either 1 or 0. So when it’s 1, it causes the values of the weights converge so fast, which also causes the error to quickly reduced.

To illustrate it, let’s assume a very simple model with only one input data x=1, one desired output y=1, and one weight, one layer. Let’s call activation function for a(x), which can be ReLU or Sigmoid. Learning rate is called *mu*.

Squared measured error and it’s derivative will be:

L(w) = (a(x) - y)^2

L’(w) = 2\*( a(x) - y) \* a’(x)

We update weights by:

w := w - mu\*L’(w)

Which is same as

w := w - mu\*2\*( a(x) - y) \* a’(x)

a’(x) determines how much the weight changes on each update.

The value of a’(x) is always 1 in ReLU, so it will update weights more quickly. But the value of a’(x) is below 1, is far below 1, so weights may not be updated so quickly.

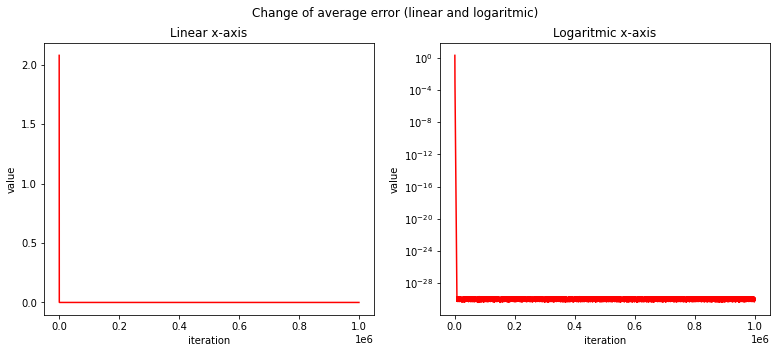
This was just to illustrate, this is not how the code in assignment works. Our case has more samples, each row has 4 columns, it has 2 layers and more weights.

### Assingment 2: **Do a search for the best learning rate. Also experiment with the necessary number of epochs. Use these values for learning rate: [0.001, 0.01, 0.1, 1, 10]. Report on your findings.**

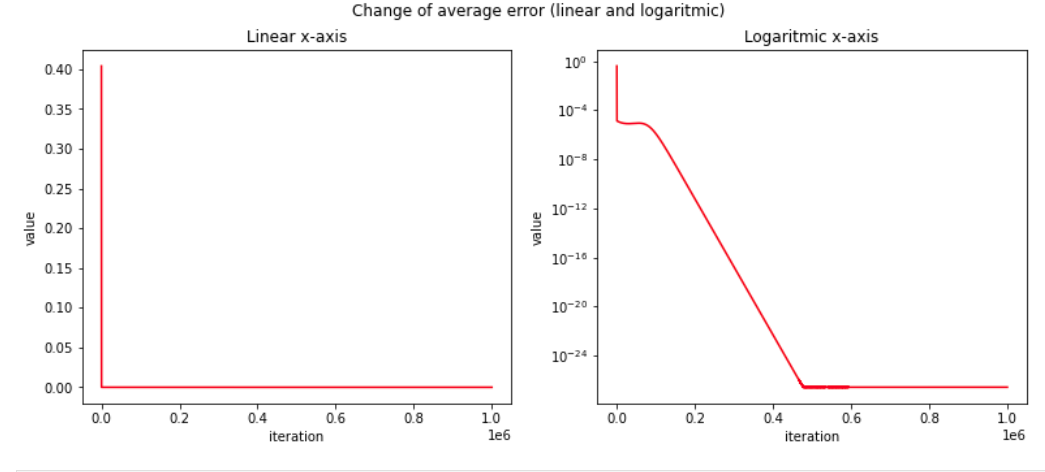
The assignment doesn’t specify if I should use ReLU, Sigmoid, or choose between them. But since we changed from ReLU to Sigmoid I assume we’re going to use Sigmoid to find the best learning rate.

When learning rate is 10, MSE quickly converges to zero. Maybe in reality it MSE doesn’t become 0, but it just reduces to a very little value. But since Python’s limit is 1e-99 it just prints zero.

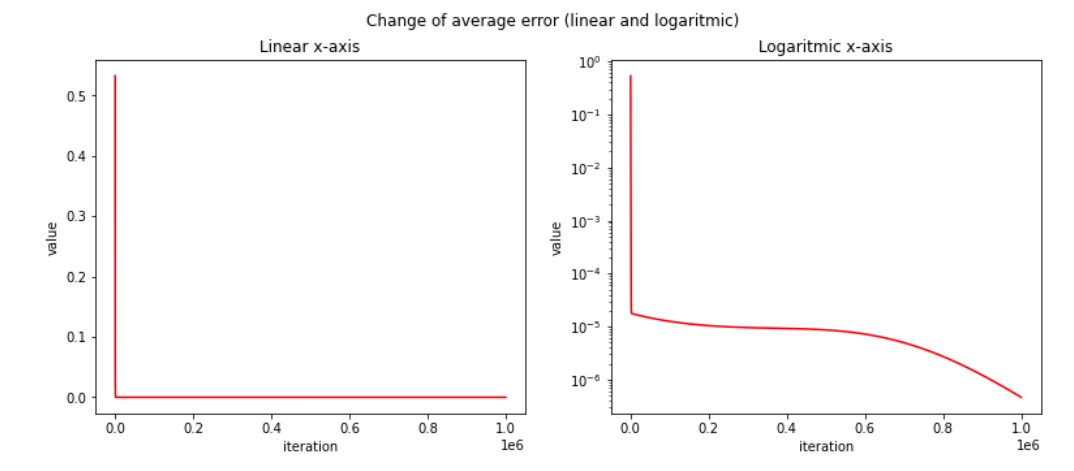
When learning rate is 1, the MSE quickly converges to a very low value. But it oscillates around a very low value.



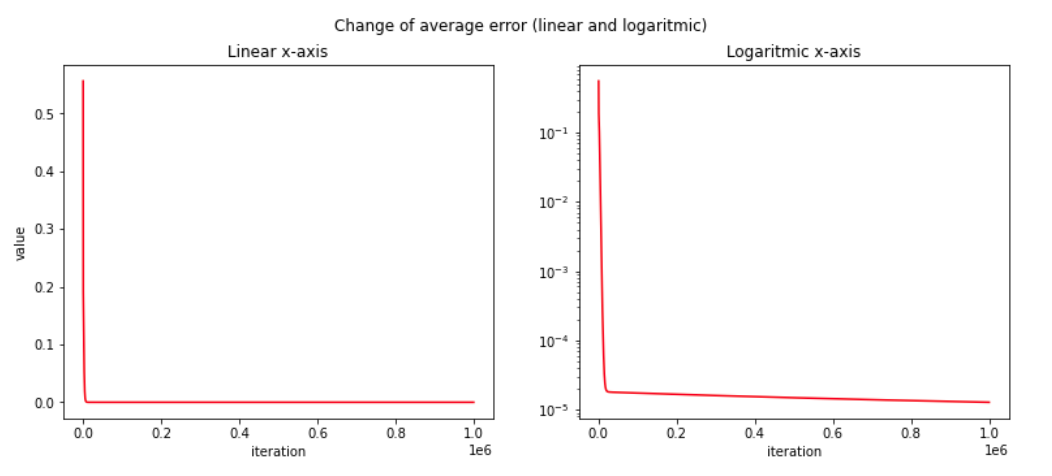
When we reduce the learning rate to 0.1 the the error is reduced more slowly, but the value it’s reduced to is higher than when learning rate is 0.1. And it’ll oscillate around a value at the end, but in logarithmic scale that’s not big.



When learning rate is 0.01, it doesn’t oscillate. It just reduces to a low value.



When learning rate is 0.001, it reduces to a low value.

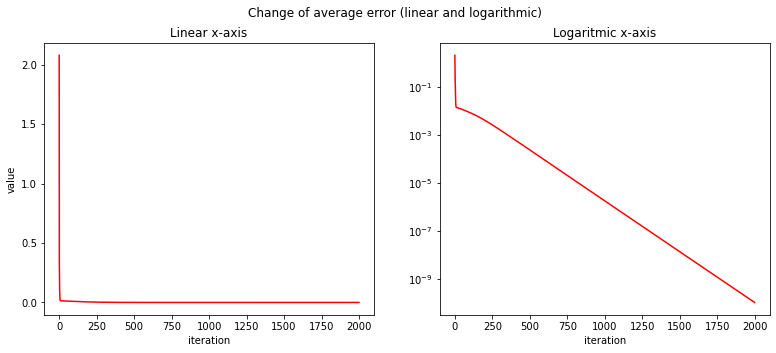


I’ll put learning rate to 1.0. It converges quickly. In this case we don’t need to fear over-learning. How the traffic light works seems to be based on very predictable rules. So we want the algorithm to follow those rules exactly. Thus learning rate of 1.0 does it fine.

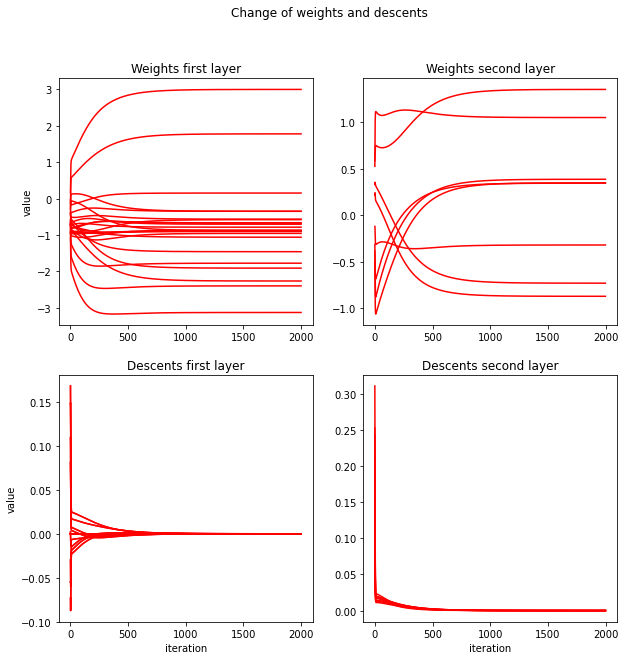
I don’t think we need 1 million epoch, since with learning rate 1.0, MSE is reducing quickly and after 2000 it’s not reducing much, and after 7000 that MSE is oscillating. So we set the epoch equal to 2000.

Also we set learning rate to 1.0, and epoch to 2000.

We see the error is reduced fast:



We see weights are quickly converging to some value:



### Assignment 3: 3: Add another “hidden” layer(hidden layers exist between the input and output layers). Choose the size of the layer yourself. Retrain the network. Here, your knowledge about learning rate and epochs from the previous task might come in handy. Is this network more or less performant than the previous version? What does this change imply?